目 录

1 实验综述

本实验实现了一个基于 numpy 和 pandas 库和基于 winequality 的 SVM 和 XGBoost 模型，实现了对 winequality 数据集的二分类任务，同时对比了 SVM 和 XGBoost 在二分类任务上的一些表现。特别地，我们还探索了两种方法的优化方法，包括 SVM 的核函数和 XGBoost 的正则化方法。

1.1 实验目标

1. 恰当处理数据集，并划分为一个二分类数据集；
2. 利用 SVM 和 XGBoost 模型对数据集进行分类；
3. 对比评估 SVM 和 XGBoost 模型的表现；
4. 探索 SVM 和 XGBoost 的优化方法。

1.2 实现思路

1. 对数据集进行探索分析，包括数据集的基本信息，特征分布，特征相关性等；
2. 实现基本的 SVM 和 XGBoost 分类器；
3. 利用 SVM 和 XGBoost 分类器对数据集进行分类；
4. 预测结果并进行分析。

1.3 实现特色与创新点

- **虽然是简易地实现，但我们依然是在对偶空间中采用的 SMO 算法求解支持向量**
- **探索了 SVM 的 Kernel Trick 并进行了一个比较明显的可视化**
- **SVM 模型预测结果对比了标准 sklearn 库并通过和 sklearn 库的差异分析了可能的问题。**
- **实现了 XGBoost 的逻辑回归模型，并探索了 XGBoost 的预测正确率伴随着树的数量的变化的变化情况。**
- **利用 graphviz 的 Digraph 实现对 XGBoost 的可视化。**
- **最重要的是我们提出了一种基于线性空间的“决策空间”解释方法，有助于理解 XGBoost 的决策树伴随训练轮次的优化过程中节点的变化。这个部分在报告的 22 页。**

1.4 数据集说明

本实验采用 winequality-red 数据集，该数据集包含 1599 个样本，每个样本包含 11 个特征，其中 10 个特征为输入特征，1 个特征为输出特征，输出特征为葡萄酒的质量，取值范围为 0-9，0 为最差，9 为最好。具体的数据集地址为： https://archive.ics.uci.edu/ml/datasets/wine+quality。

2 实验流程

2.1 数据预处理

2.1.1 数据探索

用 pandas 读取数据，并按照不同类别进行统计，发现质量集中在 5-7 之间，其中 5 和 6 的样本最多，质量为 1,2,3 和 8 的样本最少，具体的统计结果如下所示：

表 1: 样本数量统计

类别	0	1	2	3	4	5	6	7	8	9
数量	0	0	0	10	53	681	638	199	18	0

2.1.2 样本二分类化

考虑到采用 SVM 和 XGBoost 模型进行二分类问题解释性较强，实现起来也比较简单，同时应该有较好的可视化预期，我们将样本分为两类，质量小于等于 7 的样本为一类，质量大于 6 的样本为一类，这样我们就得到了一个二分类问题。

2.1.3 特征 PCA 降维

考虑到 winequality 数据集的特征数量较多，我们希望能够对特征进行降维，以便于后续的模型训练。我们采用了实验三中的 PCA 降维的方法，将原始的 10 维特征降维到 2 维，这样我们就可以直观地看到数据的分布情况。具体的降维过程如下所示：

```
#定义PCA函数，把数据降到2维
#去中心化
X -= np.mean(X,axis=0)
#计算协方差矩阵
cov = np.cov(X,rowvar=False)
#计算特征值和特征向量
eig_vals,eig_vecs = np.linalg.eig(cov)
#对特征值进行排序
eig_vals_index = np.argsort(eig_vals)[::-1]
#取前两个特征向量
eig_vecs = eig_vecs[:,eig_vals_index[:2]]
#计算降维后的数据
X_pca = np.dot(X,eig_vecs)
#把特征标准化
X_pca /= np.std(X_pca,axis=0)
```

降维之后我们以前两个主成分为横纵坐标，画出了降维后的数据的散点图，如下所示：

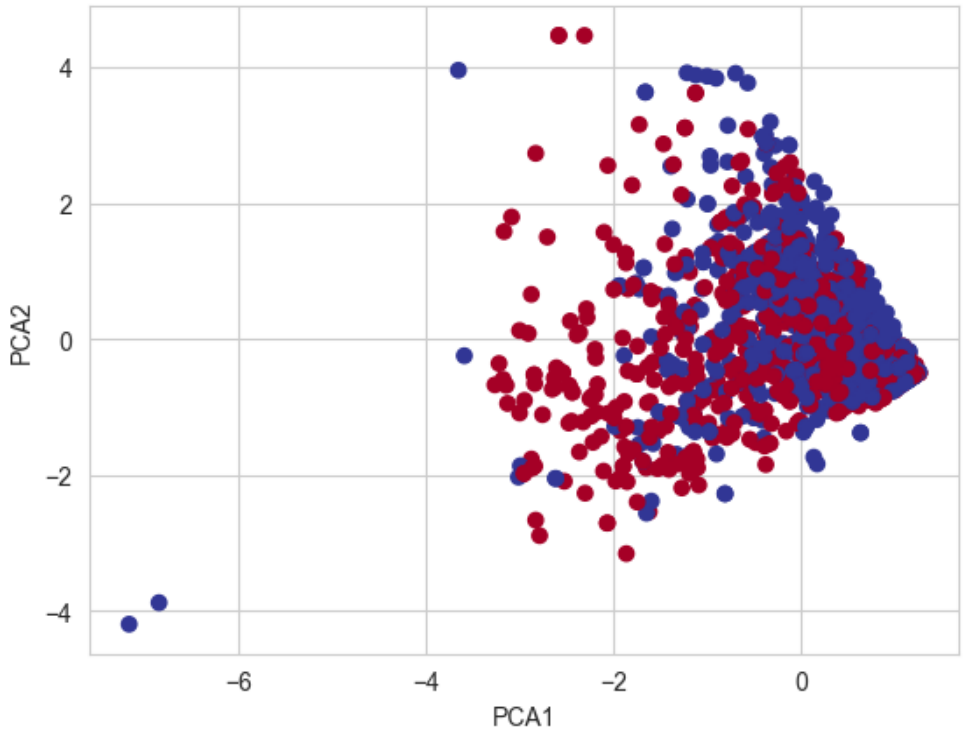


图 1: 降维后的数据散点图

从图中我们可以看出，降维后的数据分布比较不均匀，有明显的聚集现象，这一方面说明 PCA 降维的局限性，同时也说明我们需要在分类时充分考虑到数据的非线性性质。

2.2 SVM 模型建立

SVM, 支持向量机，作为一种二分类模型时，它的基本模型是定义在特征空间上的间隔最大的线性分类器，间隔最大使它有别于感知机；支持向量机还包括核技巧，这使它成为实质上的非线性分类器。

SVM 的目标在于最大化间隔，即最大化支持向量到超平面的距离，这样可以使得模型更加鲁棒，因为支持向量的位置是由数据决定的，而不是由参数决定的，考虑到软间隔的情况下公式如下：

$$\begin{aligned} &\min_{w,b,\xi} \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i \\ &s.t. \quad y_i(w^T x_i + b) \geq 1 - \xi_i \\ &\xi_i \geq 0, i = 1, 2, ..., m \end{aligned} \tag{1}$$

其中 ξ_i 是松弛变量，它允许样本出现在分隔面错误的一侧，C 是惩罚系数，它的值越大，对误分类的惩罚越大。

上面问题的拉格朗日函数为：

$$
\begin{aligned}
L(w,b,\xi,\alpha,\mu) =& \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y_i(w^T x_i + b) - 1 + \xi_i] - \sum_{i=1}^{m}\mu_i\xi_i \\
=& \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i y_i(w^T x_i + b) + \sum_{i=1}^{m}\alpha_i - \sum_{i=1}^{m}\alpha_i\xi_i - \sum_{i=1}^{m}\mu_i\xi_i \\
=& \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i y_i(w^T x_i + b) + \sum_{i=1}^{m}\alpha_i - \sum_{i=1}^{m}(\alpha_i + \mu_i)\xi_i
\end{aligned} \tag{2}
$$

求解拉格朗日函数对应的 KKT 条件为：

$$
\begin{aligned}
&\frac{\partial L}{\partial w} = w - \sum_{i=1}^{m}\alpha_i y_i x_i = 0 \\
&\frac{\partial L}{\partial b} = -\sum_{i=1}^{m}\alpha_i y_i = 0 \\
&\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \mu_i = 0 \\
&\alpha_i[y_i(w^T x_i + b) - 1 + \xi_i] = 0 \\
&\mu_i\xi_i = 0 \\
&y_i(w^T x_i + b) - 1 + \xi_i \geq 0 \\
&\alpha_i \geq 0 \\
&\mu_i \geq 0
\end{aligned} \tag{3}
$$

其中互补松弛条件和导数为 0 的条件是对偶问题的必要条件，通过求解对偶问题可以得到原始问题的解。将上面的 KKT 条件代入拉格朗日函数，得到对偶问题：

$$
\begin{aligned}
\max_{\alpha} \quad & \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j x_i^T x_j \\
s.t. \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \\
& 0 \leq \alpha_i \leq C, i = 1, 2, ..., m
\end{aligned} \tag{4}
$$

由此可见，SVM 的求解只需要计算在决策边界上的支持向量，而其他的样本都不需要计算，这样就大大减少了计算量。于是，**考虑哪些样本是支持向量就是对偶空间中求解 SVM 的核心**。同时通过讨论 α 的取值，我们可以决定哪些样本是支持向量：

1. 当 $\alpha_i = 0$ 时，$y_i(w^T x_i + b) = 1$，此时样本在分隔面正确的一侧，不需要计算；
2. 当 $0 < \alpha_i < C$ 时，$y_i(w^T x_i + b) = 1$，此时样本在分隔面上，需要计算；
3. 当 $\alpha_i = C$ 时，$y_i(w^T x_i + b) = 1$，此时样本在分隔面错误的一侧，需要计算。

不过求解上面的对偶问题也是一个凸二次规划问题，可以通过 SMO 算法来求解，步骤如下：

1. 选择两个变量 α_i 和 α_j，固定其他变量，求解对偶问题，得到最优解 α_i^* 和 α_j^*；
2. 计算 E_i 和 E_j；
3. 选择变量 α_i 和 α_j，使得 $|E_i - E_j|$ 最大；
4. 选择变量 α_i 和 α_j，固定其他变量，求解对偶问题，得到最优解 α_i^* 和 α_j^*；

一旦求得 α，就可以求得 w 和 b：

$$
\begin{aligned}
w =& \sum_{i=1}^{m} \alpha_i y_i x_i \\
b =& y_i - \sum_{i=1}^{m} \alpha_i y_i x_i^T x_j
\end{aligned}
\tag{5}
$$

这个过程相当于通过求解对偶问题，得到了原始问题的解。

进一步地对于软阈值的 SVM 问题，通过讨论 C 的取值，可以得到以下结论：

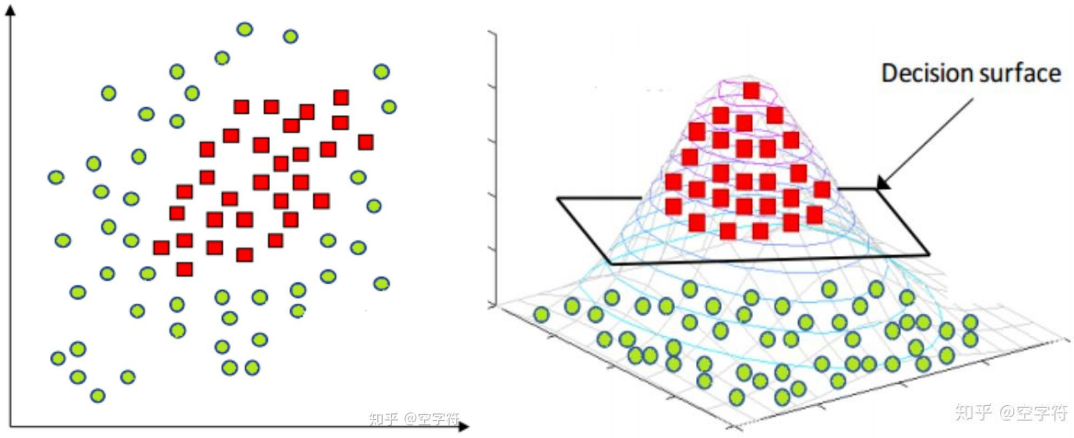
1. 当 C 趋近于无穷大时，相当于没有松弛变量，此时模型相当于一个硬间隔模型，此时模型对噪声非常敏感，容易过拟合；

2. 当 C 趋近于 0 时，相当于允许所有的样本都可以在分隔面的错误一侧，此时模型相当于一个软间隔模型，此时模型对噪声不敏感，容易欠拟合；

C 的取值需要根据具体的问题来确定，一般来说，C 的取值越大，模型的泛化能力越差，但是对训练集的拟合能力越强，反之亦然。C 的取值情况便于我们探索 SVM 对于噪声的敏感性，同时也便于我们探索 SVM 的过拟合和欠拟合的情况。

2.3 SVM 的 Kernel Trick

Kernel Trick 是 SVM 的一个重要的优化方法，它的思想是通过一个非线性的映射 ϕ 把原始的输入空间映射到一个更高维的特征空间，使得原本线性不可分的问题变成了线性可分的问题，从而使得 SVM 可以处理非线性问题。Kernel Trick 的一个简易原理如下图所示：



(a) 原始空间　　(b) 映射空间

图 2: Kernel Trick 示意图

（图片来源： https://zhuanlan.zhihu.com/p/148034340）

由图可见，原始空间中的数据是线性不可分的，但是通过一个映射 ϕ，把原始空间映射到一个更高维的特征空间，原本线性不可分的问题就可以在高维空间中使用超平面进行分割，从而使得原本线性不可分的问题变成了线性可分的问题。

同时在 SVM 模型中，我们定义 $K(x,z) = \phi(x)^T\phi(z)$，这样我们就可以通过核函数 $K(x,z)$ 来计算映射后的数据的内积，而不需要显式地计算映射函数 ϕ，这样就大大减少了计算量。

相当于在引入 Kernel Trick 之后，我们的 SVM 模型在对偶问题中的目标函数变成了：

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \\ s.t. \quad & \sum_{i=1}^{m} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C, i = 1, 2, ..., m \end{aligned} \tag{6}$$

这样我们就可以通过选择不同的核函数来处理不同的问题，常用的核函数有线性核函数、多项式核函数、高斯核函数、拉普拉斯核函数、Sigmoid 核函数等。实验中我们选择了高斯核函数和二次多项式核函数进行了探索。

2.3.1 高斯核函数

高斯核函数的定义如下：

$$K(x, z) = exp(-\frac{\|x - z\|^2}{2\sigma^2}) \tag{7}$$

高斯核函数最核心的一点是能够将数据映射到无穷维的空间，这样就能够处理非线性的数据。高斯核函数的参数 σ 控制了映射后的数据的分布，σ 越大，映射后的数据越分散，σ 越小，映射后的数据越集中。

通过泰勒展开，一个一维的标量 x 经过映射后的结果如下：

$$\phi(x) = [1, x, \frac{x^2}{2!}, \frac{x^3}{3!}, ..., \frac{x^n}{n!}, ...]exp(-\frac{x^2}{2\sigma^2}) \tag{8}$$

2.3.2 二次多项式核函数

二次多项式核函数的定义如下：

$$K(x, z) = (x^T z)^d \tag{9}$$

其中 d 是多项式的次数，d 越大，映射后的数据越分散，d 越小，映射后的数据越集中。二次即 d=2 的情况下，经过映射后的结果如下：

$$\phi(x) = [1, \sqrt{2}x_1, \sqrt{2}x_2, ..., \sqrt{2}x_n, x_1^2, x_2^2, ..., x_n^2] \tag{10}$$

2.4 SVM 类的 python 实现

我们把整个 SVM 类的 python 实现分为了以下几个部分：

- **SVM 类的初始化**：初始化 SVM 类的参数，包括 C，核函数，核函数的参数等；
- **SVM 类的 fit 函数**：SVM 类的训练函数，通过 SMO 算法求解对偶问题，得到最优解 α，进而求得 w 和 b；
- **SVM 类的 predict 函数**：SVM 类的预测函数，通过 w 和 b 计算预测结果；
- **SVM 类的 score 函数**：SVM 类的评估函数，通过预测结果和真实结果计算准确率；
- **SVM 类的 plot 函数**：SVM 类的可视化函数，通过 matplotlib 库画出 SVM 的决策边界和支持向量。

- **SVM 类的 kernel 函数**：SVM 类的核函数，包括线性核函数，高斯核函数和二次多项式核函数。
- **SVM 类的 get_support_vector 函数**：SVM 类的支持向量函数，通过 α 的取值，得到支持向量的索引。

具体结果如下：

```
import random as rnd
class SVM():
    """
        Simple implementation of a Support Vector Machine using the
        Sequential Minimal Optimization (SMO) algorithm for training.
    """
    def __init__(self, max_iter=10000, kernel_type='linear', C=1.0, epsilon=0.0001):
        self.kernels = {
            'linear' : self.kernel_linear,
            'quadratic' : self.kernel_quadratic,
            'gaussian' : self.kernel_gaussian
        }
        self.max_iter = max_iter
        self.kernel_type = kernel_type
        self.C = C
        self.epsilon = epsilon
    def fit(self, X, y):
        # Initialization
        n, d = X.shape[0], X.shape[1]
        alpha = np.zeros((n))
        kernel = self.kernels[self.kernel_type]
        count = 0
        while True:
            count += 1
            alpha_prev = np.copy(alpha)
            for j in range(0, n):
                i = self.get_rnd_int(0, n-1, j) # Get random int i~=j
                x_i, x_j, y_i, y_j = X[i,:], X[j,:], y[i], y[j]
                k_ij = kernel(x_i, x_i) + kernel(x_j, x_j) - 2 * kernel(x_i, x_j)
                if k_ij == 0:
                    continue
                alpha_prime_j, alpha_prime_i = alpha[j], alpha[i]
                (L, H) = self.compute_L_H(self.C, alpha_prime_j, alpha_prime_i, y_j, y_i)

                # Compute model parameters
                self.w = self.calc_w(alpha, y, X)
                self.b = self.calc_b(X, y, self.w)

                # Compute E_i, E_j
                E_i = self.E(x_i, y_i, self.w, self.b)
                E_j = self.E(x_j, y_j, self.w, self.b)

                # Set new alpha values
                alpha[j] = alpha_prime_j + float(y_j * (E_i - E_j))/k_ij
                alpha[j] = max(alpha[j], L)
                alpha[j] = min(alpha[j], H)

                alpha[i] = alpha_prime_i + y_i*y_j * (alpha_prime_j - alpha[j])

            # Check convergence
            diff = np.linalg.norm(alpha - alpha_prev)
```

```
            if diff < self.epsilon:
                break

            if count >= self.max_iter:
                print("Iteration number exceeded the max of %d iterations" % (self.max_iter))
                return

        # Compute final model parameters
        self.b = self.calc_b(X, y, self.w)
        if self.kernel_type == 'linear':
            self.w = self.calc_w(alpha, y, X)
        # Get support vectors
        alpha_idx = np.where(alpha > 0)[0]
        support_vectors = X[alpha_idx, :]
        return support_vectors, count,alpha_idx

    def predict(self, X):
        return self.h(X, self.w, self.b)
    def calc_b(self, X, y, w):
        b_tmp = y - np.dot(w.T, X.T)
        return np.mean(b_tmp)
    def calc_w(self, alpha, y, X):
        return np.dot(X.T, np.multiply(alpha,y))
    # Prediction
    def h(self, X, w, b):
        return np.sign(np.dot(w.T, X.T) + b).astype(int)
    # Prediction error
    def E(self, x_k, y_k, w, b):
        return self.h(x_k, w, b) - y_k
    def compute_L_H(self, C, alpha_prime_j, alpha_prime_i, y_j, y_i):
        if(y_i != y_j):
            return (max(0, alpha_prime_j - alpha_prime_i), min(C, C - alpha_prime_i +
                                                               alpha_prime_j))
        else:
            return (max(0, alpha_prime_i + alpha_prime_j - C), min(C, alpha_prime_i +
                                                               alpha_prime_j))
    def get_rnd_int(self, a,b,z):
        i = z
        cnt=0
        while i == z and cnt<1000:
            i = rnd.randint(a,b)
            cnt=cnt+1
        return i

    # Define kernels
    def kernel_linear(self, x1, x2):
        return np.dot(x1, x2.T)
    #Quadratic kernel
    def kernel_quadratic(self, x1, x2):
        return (np.dot(x1, x2.T) ** 5)
    #Gaussian kernel
    def kernel_gaussian(self, x1, x2, sigma=0.5):
        return np.exp(-np.linalg.norm(x1-x2)**2 / (2 * (sigma ** 2)))
    #Quadratic mapping
    def kernel_quadratic_mapping(self, X):
        n, d = X.shape[0], X.shape[1]
        X_map = np.zeros((n, int(d*(d+1)/2)))
        for i in range(n):
            cnt = 0
```

```
            for j in range(d):
                X_map[i][cnt] = X[i][j] ** 2
                cnt += 1
            for j in range(d):
                for k in range(j+1, d):
                    X_map[i][cnt] = np.sqrt(2) * X[i][j] * X[i][k]
                    cnt += 1
        return X_map
    #Gaussian mapping
    def kernel_gaussian_mapping(self, X, sigma=0.5):
        n, d = X.shape[0], X.shape[1]
        X_map = np.zeros((n, n))
        for i in range(n):
            for j in range(n):
                X_map[i][j] = self.kernel_gaussian(X[i], X[j], sigma)
        return X_map
    #Kernel mapping
    def mapping(self, X):
        if self.kernel_type == 'quadratic':
            return self.kernel_quadratic_mapping(X)
        elif self.kernel_type == 'gaussian':
            return self.kernel_gaussian_mapping(X)
        else:
            return X
    #print support vector indices
    def print_support_vector_idx(self, alpha_idx):
        print("Support vector indices:", alpha_idx)

    #Count number of support vectors
    def count_support_vectors(self, alpha_idx):
        return len(alpha_idx)

    #Draw results in 3D
    def plot_decision_boundary_3d(self, X,y):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.RdYlBu)

        plt.show()

    #Draw decision boundary in 2D
    def plot_decision_boundary(self, X, y):
        # Set min and max values and give it some padding
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        h = 0.01
        # Generate a grid of points with distance h between them
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
        # Predict the function value for the whole grid
        Z = self.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        # Plot the contour and training examples
        plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu)
        plt.show()
```

2.5 SVM 实验结果评估

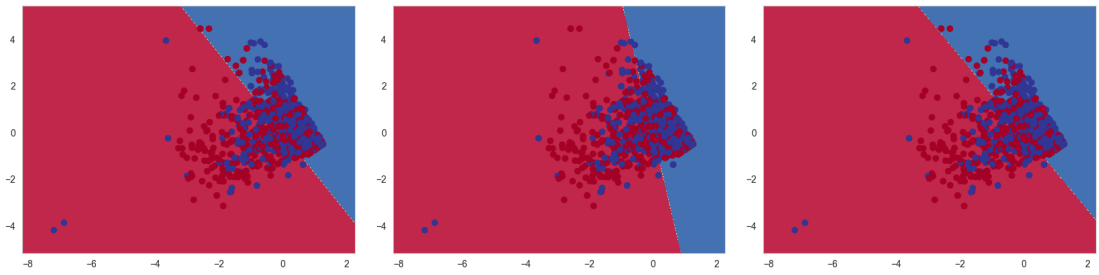
2.5.1 SVM 模型结果初步评估

基于线性核, 高斯核和二次多项式核的 SVM 模型的准确率如下表所示：

表 2: SVM 模型准确率

核函数	线性核	二次多项式核	高斯核
准确率	0.44375	0.40398	0.44688

从表中我们可以看出，三种核函数的准确率都不是很高，这可能是因为我们的数据集本身就是一个非线性的数据集，我们使用的三种核都不能很好地辨识其中的非线性特性。但相较而言高斯核还是能取得更好一些的效果，说明经过 Kernel Trick 之后，SVM 模型的效果确实有所提升。以下三张图分别是三种核函数的决策边界在原始数据集上的可视化结果，高维空间下的决策边界考虑为高维超平面在低维度下的投影：

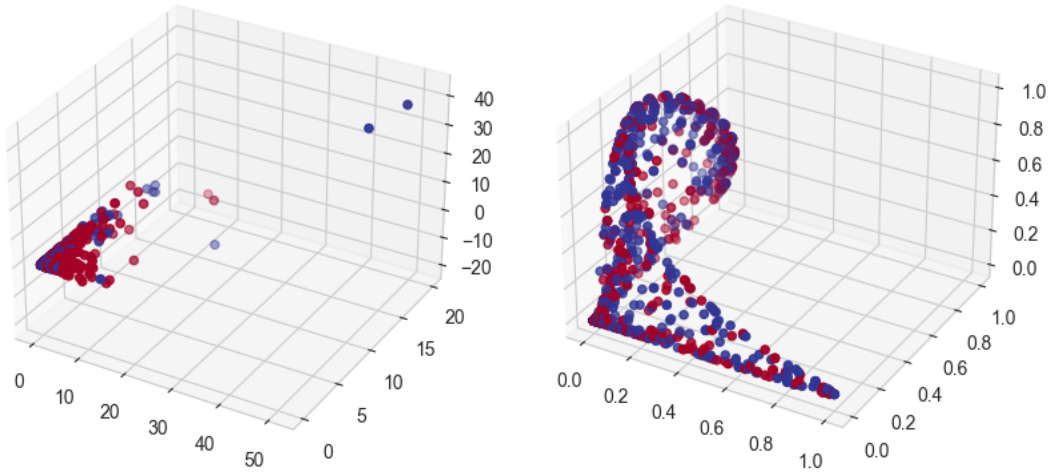


(a) 线性核 (b) 二次多项式核 (c) 高斯核

图 3: SVM 决策边界可视化

由于两种样本分布在原始空间中比较集中，所以三种核函数的线性决策边界都不能很好地分开两类数据，这也是导致准确率不高的原因之一。

为了更直观地分析 Kernel Trick 的效果，我们更直观地展示了经过核方法映射后样本数据点在高维空间的分布情况如下图所示： (我们仅能展示三维分布，即经过映射后的三个主成分分量下的样本分布)



(a) 二次多项式核　　(b) 高斯核

图 4: SVM 核方法映射后的数据分布可视化

可见基于高斯核的 Kernel Trick 映射后的数据分布的模式得到了较好的“拆散”而二次核基本保持了原本的样本模式，我们推测这是为什么高斯核能适当提升模型性能但二次核映射后模型性能下降的原因。

2.5.2 与标准库的对比

这个部分我们将对比我们实现的 SVM 类和标准库 sklearn 中的 SVM 实现的情况。下面的表格是我们实现的 SVM 类和 sklearn 中的 SVM 类在不同核函数下的准确率对比：

表 3: SVM 模型准确率对比

核函数	线性核	二次多项式核	高斯核
我们实现的 SVM 类	0.44375	0.40398	0.44688
sklearn 中的 SVM 类	0.65	0.6	0.6375

可见我们的模型和标准库中的方法还是有较大差距的，但考虑到 SVM 模型本身作为一种比较复杂的模型，在求解方法或者参数上可能还有很大的优化空间，因此这种差距也是可以理解的。

2.6 XGBoost 模型建立

XGBoost 是一种基于决策树的集成学习模型，它的全称是 eXtreme Gradient Boosting，它是一种 Boosting 算法，它的核心思想是通过训练多个弱分类器，然后将这些弱分类器进行线性组合，得到一个强分类器，从而提升模型的性能。我们知道 XGBoost 是一个由 k 个基模型组成的一个加法算式即：

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i) \tag{11}$$

其中 f_k 是基模型，K 是基模型的个数，$\hat{y}_i$ 是最终的预测结果。损失函数即：

$$Obj(\theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{12}$$

其中 $\Omega(f_k)$ 是正则化项，它的作用是防止过拟合，同时也可以用来控制模型的复杂度，$l(y_i, \hat{y}_i)$ 是损失函数，它的作用是衡量模型的拟合程度，常用的损失函数有平方损失函数、绝对损失函数、对数损失函数等。

由于 Boosting 模型是一个前向分步加法模型，即：

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i) \tag{13}$$

于是目标函数可以写成：

$$Obj(\theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) \tag{14}$$

求解此时的目标函数，就是求解基模型 f_t。考虑到可以把 f_t 看成新一步的增量，所以我们根据泰勒展开，可以得到：

$$Obj(\theta) \approx \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{15}$$

由于在第 t 次迭代时，前 t-1 次迭代的结果 $\hat{y}_i^{(t-1)}$ 是已知的，所以我们可以把它看成常数，于是目标函数可以写成：

$$Obj(\theta) \approx \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{16}$$

所以我们只需要求解每一步的损失函数的一阶导数 g_i **和二阶导数** h_i**，再根据加法模型就可以求解基模型** f_t。总的来看，XGBoost 把每一步的目标函数分解成了两部分，一部分是损失函数的一阶导数，一部分是损失函数的二阶导数，这样就大大简化了模型的求解过程。

2.7 基于决策树的 XGBoost 模型

XGBoost 模型的基模型一般是决策树。我们定义决策树为：

$$f_t(x) = w_{q(x)} \tag{17}$$

其中 $q(x)$ 是决策树的结构，即样本在决策树中的路径，w 是决策树的叶子节点的权重。所以 $w_{q(x)}$ 就是样本在决策树中的路径对应的叶子节点的权重。决策树的复杂度可以定义为：

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{18}$$

其中 T 是决策树的叶子节点的个数，γ 和 λ 是正则化项的参数，用来控制决策树的复杂度。**把决策树的复杂度加入到目标函数中，就可以得到最终的目标函数：**

$$Obj(\theta) = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{19}$$

为简化模型，我们定义：

$$G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i \tag{20}$$

其中 I_j 是样本在决策树中的第 j 个叶子节点的索引，G_j 是样本在决策树中的第 j 个叶子节点的一阶导数的和，H_j 是样本在决策树中的第 j 个叶子节点的二阶导数的和。

于是目标函数可以写成：

$$Obj(\theta) = \sum_{j=1}^{T}[G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2] + \gamma T \tag{21}$$

由于 G_j 和 H_j 是前 t-1 次迭代的结果，所以它们是已知的，所以我们可以把它们看成常数，只有 w_j 是未知的，所以我们可以对 w_j 求导，得到：

$$\frac{\partial Obj}{\partial w_j} = G_j + (H_j + \lambda)w_j \tag{22}$$

令上式等于 0，可以得到：

$$w_j^* = -\frac{G_j}{H_j + \lambda} \tag{23}$$

所以最终的目标函数可以写成：

$$Obj(\theta) = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T \tag{24}$$

2.8 XGBoost 分裂节点的贪心算法

XGBoost 模型的核心是分裂节点的贪心算法，它的核心思想是：对于每一个节点，我们计算分裂后的目标函数的增益，然后选择增益最大的节点进行分裂，直到满足停止条件为止。其步骤如下：

- **第一步：计算分裂后的目标函数的增益**

  对于每一个节点，我们计算分裂后的目标函数的增益，即：

  $$Gain = \frac{1}{2}[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}] - \gamma \tag{25}$$

  其中 G_L 和 G_R 是分裂后左右两个节点的一阶导数的和，H_L 和 H_R 是分裂后左右两个节点的二阶导数的和，λ 和 γ 是正则化项的参数。

- **第二步：选择增益最大的节点进行分裂**

  选择增益最大的节点进行分裂，直到满足停止条件为止。

当然，为了防止过拟合，我们还需要定义停止条件，一般来说，我们可以定义以下几种停止条件：

- **节点的深度达到最大深度**

  一般来说，我们可以定义树的最大深度，当节点的深度达到最大深度时，就停止分裂。

- **节点的样本数达到最小样本数**

  一般来说，我们可以定义节点的最小样本数，当节点的样本数达到最小样本数时，就停止分裂。

- **节点的增益达到最小增益**

  一般来说，我们可以定义节点的最小增益，当节点的增益达到最小增益时，就停止分裂。

贪心算法的伪代码如下：

Algorithm 1 XGBoost 分裂节点的贪心算法

Require: 训练集 $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$，正则化项的参数 λ 和 γ，最大深度 max_depth，最小样本数 $min\_samples\_split$，最小增益 min_gain
Ensure: 决策树的结构 $q(x)$ 和叶子节点的权重 w

function SPLIT($D, \lambda, \gamma, max\_depth, min\_samples\_split, min\_gain$)
    初始化根节点 $root$
    $Q \leftarrow \{root\}$
    while Q 不为空 **do**
        从 Q 中取出一个节点 $node$
        从 $node$ 中取出节点的样本 D_{node}
        从 $node$ 中取出节点的深度 $depth$
        从 $node$ 中取出节点的增益 $gain$
        if $depth < max\_depth$ 且 $|D_{node}| > min\_samples\_split$ 且 $gain > min\_gain$ **then**
            计算 D_{node} 的一阶导数的和 G_{node}
            计算 D_{node} 的二阶导数的和 H_{node}
            计算 D_{node} 的增益 $gain$
            选择增益最大的特征 f 和最优的分割点 s
            根据特征 f 和分割点 s 分裂 $node$，得到左节点 $node_L$ 和右节点 $node_R$
            把 $node_L$ 和 $node_R$ 加入到 Q 中
        end if
    end while
    return $root$
end function

2.9 XGBoost 模型的 python 实现

我们把整个 XGBoost 模型的 python 实现分为了以下几个部分：

- **XGBoost 类的初始化**：初始化 XGBoost 类的参数，包括正则化项的参数 λ 和 γ，最大深度 max_depth，最小样本数 $min\_samples\_split$，最小增益 min_gain 等；
- **XGBoost 类的 cart_split 函数**：XGBoost 类的分裂节点的函数，通过贪心算法求解决策树的结构 $q(x)$ 和叶子节点的权重 w；
- **XGBoost 类的 fit 函数**：XGBoost 类的训练函数，通过贪心算法求解决策树的结构 $q(x)$ 和叶子节点的权重 w；
- **XGBoost 类的 predict 函数**：XGBoost 类的预测函数，通过决策树的结构 $q(x)$ 和叶子节点的权重 w 计算预测结果；
- **基于 graphviz 库的 XGBoost 类的 plot 函数**：XGBoost 类的可视化函数，通过 graphviz 库画出决策树的结构 $q(x)$。

我们的 XGBoost 类的核心是一个包含了决策树结构和叶子节点权重的字典，我们把它称为 $tree_structure$, 我们基于 cart_split 函数求解 $tree\_structure$，然后基于 $tree\_structure$ 求解预测结果。

特别有必要说明的一点是，我们在实现 XGBoost 模型时考虑到了 Linear 和 Logistic 两种情况，所以我们的 XGBoost 类的初始化函数中有一个参数是 $objective$，用来指定是 Linear 还是 Logistic，如果是 Linear，就用平方损失函数，如果是 Logistic，就用对数损失函数。

引入 Logistic 的原因是，基于 Logistic 的 XGBoost 模型求解的是一个概率维度的二分类问题。把数据映射到 0-1 区间符合我们原始定义的二分类标签。

具体代码如下：

```
#定义XGB类
#该XGB类通过递归的方式构造XGB中的Cart树，分裂的依据是分裂后的增益最大
class XGB:

    def __init__(self,
                 base_score=0.5,
                 max_depth=3,
                 n_estimators=10,
                 learning_rate=0.1,
                 reg_lambda=1,
                 gamma=0,
                 min_child_sample=None,
                 min_child_weight=1,
                 objective='linear'):

        self.base_score = base_score  # 最开始时给叶子节点权重所赋的值，默认0.5，
        self.max_depth = max_depth  # 最大数深度
        self.n_estimators = n_estimators  # 树的个数
        self.learning_rate = learning_rate  # 学习率，这里是每棵树要乘以的权重系数
        self.reg_lambda = reg_lambda  # L2正则项的权重系数
        self.gamma = gamma  # 正则项中，叶子节点数T的权重系数
        self.min_child_sample = min_child_sample  # 每个叶子节点的样本数（自己加的）
        self.min_child_weight = min_child_weight  # 每个叶子节点的Hessian矩阵和，下面代码会细讲
        self.objective = objective  # 目标函数，可选linear和logistic
        self.tree_structure = {}  # 用一个字典来存储每一颗树的树结构

    def xgb_cart_tree(self, X, w, m_dpth):
        '''
        递归的方式构造XGB中的Cart树
        X：训练数据集
        w：每个样本的权重值，递归赋值
        m_dpth：树的深度
        '''

        # 边界条件：递归到指定最大深度后，跳出
        if m_dpth > self.max_depth:
            return

        best_var, best_cut = None, None
        # 这里增益的初值一定要设置为0，相当于对树做剪枝，即如果算出的增益小于0则不做分裂
        max_gain = 0
        G_left_best, G_right_best, H_left_best, H_right_best = 0, 0, 0, 0
        # 遍历每个变量的每个切点，寻找分裂增益gain最大的切点并记录下来
        for item in [x for x in X.columns if x not in ['g', 'h', 'y']]:
            for cut in list(set(X[item])):

                # 这里如果指定了min_child_sample则限制分裂后叶子节点的样本数都不能小于指定值
                if self.min_child_sample:
                    if (X.loc[X[item] < cut].shape[0] < self.min_child_sample) \
                            | (X.loc[X[item] >= cut].shape[0] < self.min_child_sample):
                        continue

                G_left = X.loc[X[item] < cut, 'g'].sum()
                G_right = X.loc[X[item] >= cut, 'g'].sum()
                H_left = X.loc[X[item] < cut, 'h'].sum()
```

```
                H_right = X.loc[X[item] >= cut, 'h'].sum()

                # min_child_weight在这里起作用，指的是每个叶子节点上的H，即目标函数二阶导的加和
                # 当目标函数为linear，即1/2*(y-y_hat)**2时，它的二阶导是1，那min_child_weight就等价于min_child_sample
                # 当目标函数为logistic，其二阶导为sigmoid(y_hat)*(1-sigmoid(y_hat))，可理解为叶子节点的纯度
                if self.min_child_weight:
                    if (H_left < self.min_child_weight) | (H_right < self.min_child_weight):
                        continue

                gain = G_left ** 2 / (H_left + self.reg_lambda) + \
                       G_right ** 2 / (H_right + self.reg_lambda) - \
                       (G_left + G_right) ** 2 / (H_left + H_right + self.reg_lambda)
                gain = gain / 2 - self.gamma
                if gain > max_gain:
                    best_var, best_cut = item, cut
                    max_gain = gain
                    G_left_best, G_right_best, H_left_best, H_right_best = G_left, G_right, H_left, H_right

        # 如果遍历完找不到可分列的点，则返回None
        if best_var is None:
            return None

        # 给每个叶子节点上的样本分别赋上相应的权重值
        id_left = X.loc[X[best_var] < best_cut].index.tolist()
        w_left = - G_left_best / (H_left_best + self.reg_lambda)

        id_right = X.loc[X[best_var] >= best_cut].index.tolist()
        w_right = - G_right_best / (H_right_best + self.reg_lambda)

        w[id_left] = int(w_left)
        w[id_right] =int(w_right)

        # 把树的结构给存下来
        tree_structure = {(best_var, best_cut): {}}
        tree_structure[(best_var, best_cut)][('left', w_left)] = self.xgb_cart_tree(X.loc[id_left], w, m_dpth + 1)
        tree_structure[(best_var, best_cut)][('right', w_right)] = self.xgb_cart_tree(X.loc[id_right], w, m_dpth + 1)

        return tree_structure

    def _grad(self, y_hat, Y):
        '''
        计算目标函数的一阶导
        '''

        if self.objective == 'logistic':
            y_hat = 1.0 / (1.0 + np.exp(-y_hat))
            return y_hat - Y
        elif self.objective == 'linear':
            return y_hat - Y
        else:
            raise KeyError('objective must be linear or logistic!')

    def _hess(self, y_hat, Y):
        '''
```

```
        计算目标函数的二阶导
        '''

        if self.objective == 'logistic':
            y_hat = 1.0 / (1.0 + np.exp(-y_hat))
            return y_hat * (1.0 - y_hat)
        elif self.objective == 'linear':
            return np.array([1] * Y.shape[0])
        else:
            raise KeyError('objective must be linear or logistic!')

    def fit(self, X: pd.DataFrame, Y):
        '''
        根据训练数据集X和Y训练出树结构和权重
        '''

        if X.shape[0] != Y.shape[0]:
            raise ValueError('X and Y must have the same length!')

        X = X.reset_index(drop='True')
        Y = Y.values
        # 这里根据base_score参数设定权重初始值
        y_hat = np.array([self.base_score] * Y.shape[0])
        for t in range(self.n_estimators):
            print('fitting tree {}...'.format(t + 1))

            X['g'] = self._grad(y_hat, Y)
            X['h'] = self._hess(y_hat, Y)

            f_t = pd.Series([0] * Y.shape[0])
            self.tree_structure[t + 1] = self.xgb_cart_tree(X, f_t, 1)

            y_hat = y_hat + self.learning_rate * f_t

            print('tree {} fit done!'.format(t + 1))

        print(self.tree_structure)

    def _get_tree_node_w(self, X, tree, w):
        '''
        以递归的方法，把树结构解构出来，把权重值赋到w上面
        '''

        if not tree is None:
            k = list(tree.keys())[0]
            var, cut = k[0], k[1]
            X_left = X.loc[X[var] < cut]
            id_left = X_left.index.tolist()
            X_right = X.loc[X[var] >= cut]
            id_right = X_right.index.tolist()
            for kk in tree[k].keys():
                if kk[0] == 'left':
                    tree_left = tree[k][kk]
                    w[id_left] = int(kk[1])
                elif kk[0] == 'right':
                    tree_right = tree[k][kk]
                    w[id_right] = int(kk[1])

            self._get_tree_node_w(X_left, tree_left, w)
```

```
            self._get_tree_node_w(X_right, tree_right, w)

    def predict_raw(self, X: pd.DataFrame):
        '''
        根据训练结果预测
        返回原始预测值
        '''

        X = X.reset_index(drop='True')
        Y = pd.Series([self.base_score] * X.shape[0])

        for t in range(self.n_estimators):
            tree = self.tree_structure[t + 1]
            y_t = pd.Series([0] * X.shape[0])
            self._get_tree_node_w(X, tree, y_t)
            Y = Y + self.learning_rate * y_t

        return Y

    def predict_prob(self, X: pd.DataFrame):
        '''
        当指定objective为logistic时，输出概率要做一个logistic转换
        '''

        Y = self.predict_raw(X)
        sigmoid = lambda x: 1 / (1 + np.exp(-x))
        Y = Y.apply(sigmoid)
        return Y
    #利用graphviz的Digraph和递归的方法绘制树结构，id为树的编号
    def plot_decision_tree(self,id=1):
        tree = self.tree_structure[id]
        dot = Digraph(comment='The Decision Tree')
        dot.node('root', 'root')
        self._plot_decision_tree(tree, dot, 'root')
        graph = pydotplus.graph_from_dot_data(dot.source)
        graph.write_png('tree.png')
        dot.view()

    def _plot_decision_tree(self, tree, dot, root):
        if not tree is None:
            k = list(tree.keys())[0]
            var, cut = k[0], k[1]
            dot.node(str(k), str(k))
            dot.edge(root, str(k))
            for kk in tree[k].keys():
                if kk[0] == 'left':
                    tree_left = tree[k][kk]
                    dot.node(str(kk), str(kk))
                    dot.edge(str(k), str(kk))
                elif kk[0] == 'right':
                    tree_right = tree[k][kk]
                    dot.node(str(kk), str(kk))
                    dot.edge(str(k), str(kk))
                    #随机设定颜色
            color = lambda: np.random.randint(0, 255)#随机生成0-255的整数
            dot.node(str(k), str(k), color='#%02X%02X%02X' % (color(), color(), color()))
               #设置节点颜色
            self._plot_decision_tree(tree_left, dot, str(kk))
            self._plot_decision_tree(tree_right, dot, str(kk))
```

2.10 XGBoost 模型的预测结果

我们分别构造了 3，5，10，12，15，20，25 棵树的 XGBoost 模型，然后对于每个模型，分别 Logistic 两种目标函数进行了训练，之后调用产生的决策树对测试集进行了预测，最后计算了预测的准确率。下面的表格是我们的结果：

表 4: XGBoost 模型准确率对比

树的个数	3	5	10	12	15	20	25
预测准确率	0.4169	0.7429	0.8809	0.8777	0.8807	0.8746	0.8746

可见构造初期随着树的个数的增加，模型的准确率也在增加，这是符合我们的预期的。**但是随着树的个数的增加，模型的准确率趋于固定并且准确率相对于树的数量为 10 左右时有一定下降，这说明模型可能产生了过拟合。**

2.11 XGBoost 模型的可视化分析

在模型原理部分，我们已经知道 XGBoost 本质上是一个基于梯度提升的前向加法集成学习模型。每一轮的迭代和梯度都基于上一轮产生的损失函数。因此我们有理由相信，不同轮次构造的决策树的结构是不同的，我们可以通过可视化的方式来分析不同轮次构造的决策树的结构的差异，从而进一步分析模型的性能。我们仅以树的个数为 10 的情况绘制其中第 1 轮，第 5 轮，第 10 轮的决策树的结构，如下图所示：

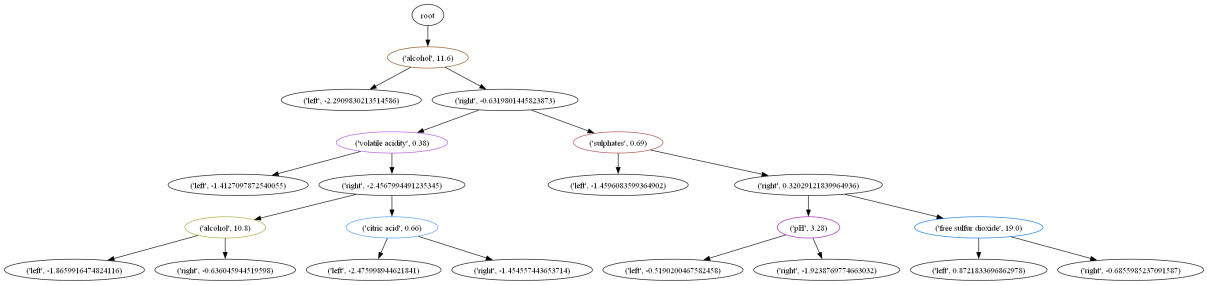


图 5: 第 1 轮决策树的结构

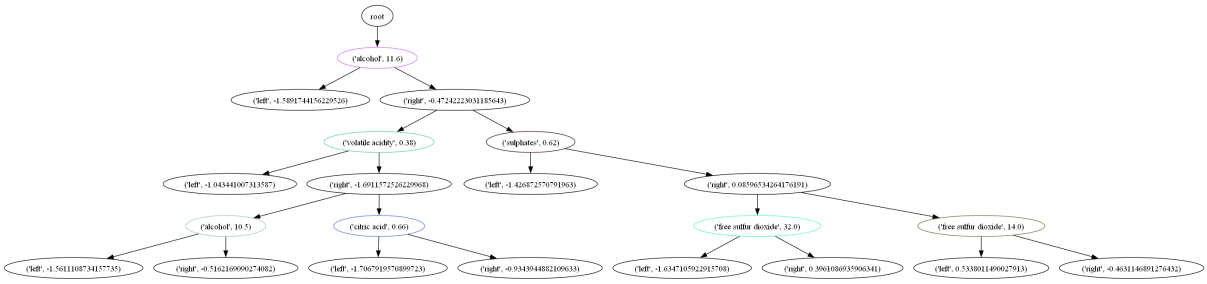


图 6: 第 5 轮决策树的结构

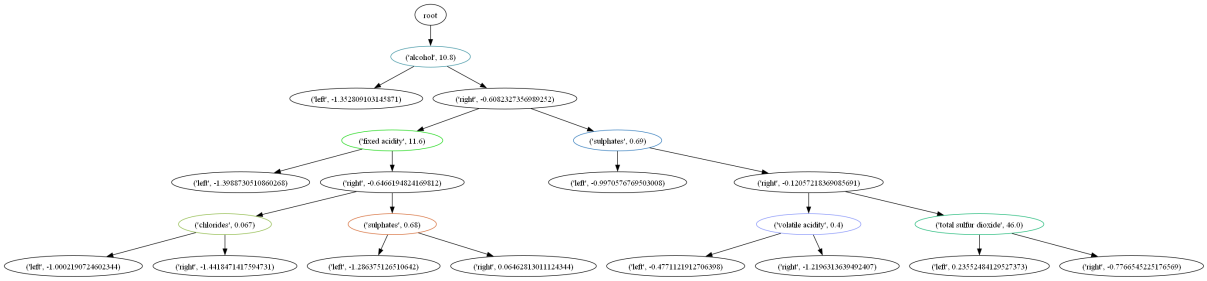


图 7: 第 10 轮决策树的结构

通过观察对比三个轮次的决策树模型我们能总结出以下两个重要的结论：

- **随着轮次的增加，上层的决策节点类型几乎不变，但决策分界线发生了变化。**
- **随着轮次的增加，下层的决策节点类型和分界线都发生了变化。**

这个现象产生是合理的，因为我们知道，上层的分裂节点代表着决策初期最有价值的决策信息增益，而下层的决策节点依赖于已经被上层决定的分类结果，在上层分界线发生变化的情况下，下层的决策节点也会发生变化。

这里我们做一个假设：**每一个决策节点都是“决策空间”中的一个“决策基”。节点类型代表了决策基的方向，而分界线代表了决策基的大小。**

在基于信息增益分裂节点的假设下：**我们可以认为上层节点的特性可以作为分类模型空间中最重要的“决策基”。**如果我们把前向加法梯度提升的过程看成是不断求解最优决策基的过程，那么我们可以认为，**随着轮次的增加，作为“主分量”的上层“决策基”的分量方向几乎不变，但是分量的大小会发生变化。**而下层的决策基则是在上层决策基的基础上进行了一定的“微调”。**由于下层分量的模长较小，故而在优化过程中大小和方向都可能会发生变化。**

2.12 与 XGBoost 库的对比

作为一个优秀的集成学习方法：XGBoost 有着独立于 sklearn 的 XGBoost 库。我们采用 XGBoost 标准库中的方法再一次进行实验：以下的表格展示决策树个数为 3，5，10 时我们的模型和标准库的模型的准确率对比：

表 5: XGBoost 模型准确率对比

树的个数	3	5	10
我们的模型	0.4169	0.7429	0.8809
标准库的模型	0.8958	0.8958	0.9083

对比可以知道，我们的模型在初期与标准库的方法性能上差距显著，但是随着树的个数的增加，我们的模型的性能逐渐接近标准库的模型的性能。这说明我们的模型的性能是可以接受的。当然标准库提供了更多探索的可能性，比如定量化特征重要性如下图：

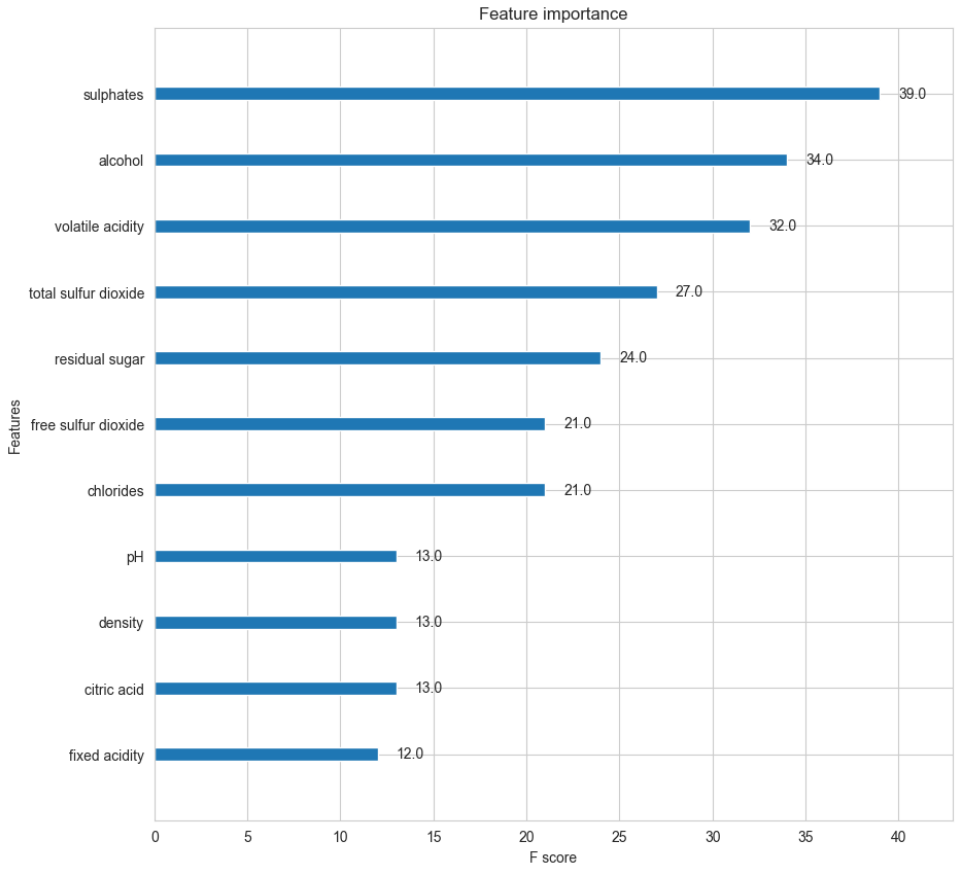


图 8: 特征重要性

我们同样可以绘制出标准库的决策树的结构如下图所示：

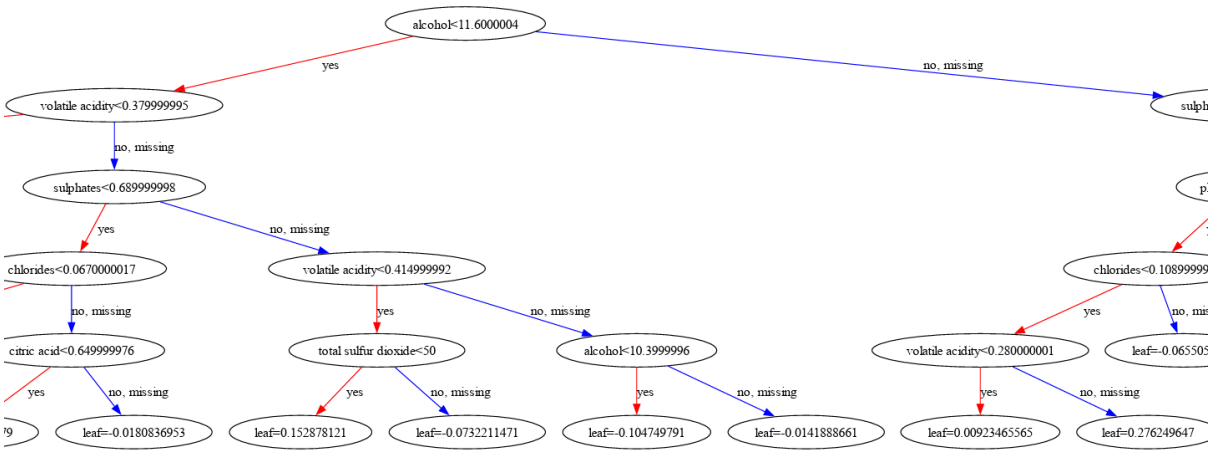


图 9: 标准库的决策树的结构

图片较大不便于展示我们仅截取其中的一部分，完整的内容我们保存到了 tree.pdf 文件中。由图我们也可以发现：**标准库的决策树的结构和我们的决策树的结构有很大的相似性。尤其是上层节点在类型和取值上几乎一致。不过标准库分裂出的决策树在下层明显更为复杂和精细。**我们认为这主要是由于分裂的控制条件不同和分裂的策略不同导致的。

不过 XGBoost 模型本质作为一种基于信息增益梯度提升的集成学习模型是一致的。上层重

要节点的相似性进一步说明了我们基于“决策基”的解释有一定的可行性。

3 SVM 和 XGBoost 的对比分析

不论从我们实现的结果还是标准库方法的结果来看 SVM 模型的分类效果在这个 winequality-red 数据集上都与 XGBoost 有较大差距。这是因为 SVM 模型的核心思想是：**通过一个超平面把数据集分成两个部分，使得两个部分的间隔最大**。而 XGBoost 模型的核心思想是：**通过不断的迭代，构造出一系列的决策树，使得每一棵决策树都能够对数据集进行更好的分类。**

SVM 模型是一种基于几何距离的分类模型，而 XGBoost 模型是一种基于信息增益的分类模型。这两种模型的核心思想是不同的，所以它们的分类效果也是不同的。

更本质地来说，SVM 模型的基础模型是一种线性分类模型，而 XGBoost 模型从开始就是一种非线性分类模型。即便我们加入了 Kernel Trick，SVM 的“非线性化”过程还是一种“缺乏引导”的，但是 XGBoost 一方面具有集成学习梯度提升的优势，另一方面每一轮的迭代都有过去历史信息作为“参考”所以我们能看到在 XGBoost 模型中随着决策树数量的上升预测的准确率快速升高（在我们实现的模型中这一点非常明显）。

从结果来看，在 3 棵树的情况下，XGBoost 模型的准确率还和 SVM 模型的准确率接近，但到有 5 棵树的情况下，XGBoost 模型的准确率就大幅度超过了 SVM 模型的准确率。这说明 XGBoost 模型的非线性化过程是有效的，而且 XGBoost 模型的集成学习梯度提升的优势也是有效的。

另外从我们在 SVM 实验中加入了 PCA 降维改变了数据集的特征分布，而在 XGBoost 实验中我们没有加入 PCA 降维。这可能也是导致两个模型分类效果存在差距的原因之一。

4 实验总结和可能的改进点

本次实验实现了一个基于数据集 winequality 的线性回归用于分类的模型，实现了 SVM 模型和 XGBoost 模型的 python 实现。

针对 SVM 模型，我们详细推导了原问题空间向对偶问题空间的转化，同时我们探索了 Kernel Trick 的原理，实现了线性核, 二次核和高斯核的 python 实现，并对比分析了三种核的分类效果。并实现了比较好的决策边界可视化和映射后的特征空间可视化。

针对 XGBoost 模型，我们详细推导了 XGBoost 的前向加法梯度提升过程，以及在每一轮迭代中如何通过贪心算法求解决策树的结构和叶子节点的权重。

特别地，通过研究**梯度提升集成树的数量和最终预测准确率的定性关系，并通过其中的决策树的结构的可视化分析，我们得出了一个基于“决策基”的解释，认为随着迭代轮次的增加，上层的决策节点类型几乎不变，但决策分界线发生了变化。而下层的决策节点类型和分界线都发生了变化。**这个现象产生是合理的，因为我们知道，上层的分裂节点代表着决策初期最有价值的决策信息增益，而下层的决策节点依赖于已经被上层决定的分类结果，在上层分界线发生变化的情况下，下层的决策节点也会发生变化。

最后通过对比 SVM 和 XGBoost 模型的预测效果差异，我们得出结论：**在 winequality-red 这样的非线性数据集上，XGBoost 模型的分类效果要优于 SVM 模型**。我们认为这是由于 **SVM 模型本质的线性特性和用 Kernel 进行“非线性化”过程中的“非引导”性，而 XGBoost 本质的非线性特性和基于信息增益梯度提升的微调过程可以快速地提升模型的分类效果。**

通过对比与标准库实现的差异，我们也发现我们模型仍然存在较大的改进空间，**比如我们可以考虑加入 PCA 降维，或者考虑加入更多的核函数，或者考虑加入更多的正则化项，或者考虑加入更多的停止条件，或者考虑加入更多的决策树的分裂策略等等。**

但总的来说我们比较完整地从数学原理和 python 实现的角度实现了 SVM 和 XGBoost 模型，对于这两个模型的理解也更加深入了。

5 参考文献和网址

[1] B. Scholkopf, A. J. Smola, K. Muller. Nonlinear component analysis as a kernel eigenvalue problem. Neural Computation 10 (5), 1299–1399, 1998.

[2] 李航，统计学习方法，清华大学出版社，2012 年.

[3] 阿泽. 知乎.【机器学习】决策树（下）——XGBoost、LightGBM（非常详细）. https://zhuanlan.zhihu.com/p/87885678.

[4] 李青峰. 知乎. 第九章 xgboost 算法-可视化与模型保存. https://zhuanlan.zhihu.com/p/99173424.